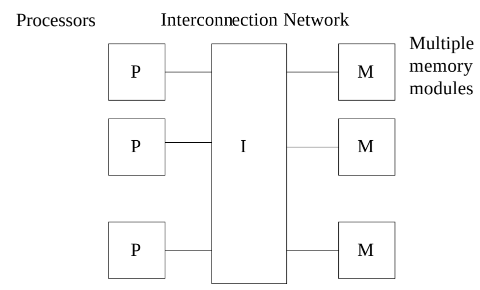
Shared-Memory Parallel Programming

Reading: PP Chapter 8

Recall the Long-Lost Shared-Memory Architecture



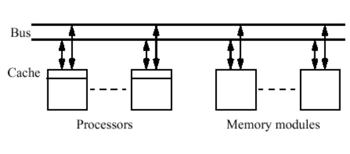
Shared-Memory Architecture also known as

- SMP (Symmetric Multiprocessor) since the view looks the same from all processors.
- UMA (Uniform Memory Access)

Recall Advantages & Disadvantages of Shared-Memory

+ All data in one address space; don't have to worry about distributing
– Not scalable, since interconnection network will either
  – saturate or
  – latency will increase

Bus Architecture with Caching



NUMA

- Technically, an architecture with caches is like a Non-Uniform Memory Access machine (local=cache, vs. global=memory).
- However, this usually reserved for the case where the local/global distinction is programmed for explicitly.

Cache Coherency Problem

- Caches keep local-to-processor copies of data in the shared memory
- If a processor modifies cached data, the data in the shared memory is no longer valid
- Worse, copies of the data in other processors' caches is invalid

Concepts for Cache Coherency

- **Invalidation**:
  - Each cache line (group of words) has a validity bit.
  - If a processor writes a word in cache, other processors' caches are checked to see if the corresponding line is present (called "snooping").
  - If the line is present, it is marked invalid.
  - The line will need to be refetched from memory if needed.

Concepts for Cache Coherency

- The alternative to snooping is to **broadcast** the written word to all processors.
- **Broadcast** is expensive because it uses bandwidth even if the processor does not have the line cached.
- **Directory-based** system is another approach: The directory knows where all copies are; sends updates selectively.

Write-Through, Write-Back, etc.

- **Write-through**: When a new value is written to a cached word, the value is immediately written to memory as well.
- **Write-back**: When a new value is written to a cached word, the value is not written to memory until the cache line is replaced with some other set of words.
- **No-Write**: Only reads are cached

Tradeoffs?

- Write-through: Memory is always up-to-date.
- Write-back: Less traffic writing stuff to memory (that might not be used between writes).
- No-Write: Typically most accesses are reads, so this achieves performance with simplicity.

MESI States for Cache Lines

- **Exclusive Modified** (M): not shared by other caches and contains modified information, i.e. main memory does not contain the current value.
- **Exclusive Unmodified** (E): not shared and was not modified.
- **Shared Unmodified** (S): unmodified and present in other caches.
- **Invalid** (I): invalid as other caches or main memory contain a modified version.

Effect on Cache Miss for Line

- Prior to reading new data:
  - **Exclusive Modified** (M): Data must be written (if not already written back)
  - **Exclusive Unmodified** (E): NOOP
  - **Shared Unmodified** (S): NOOP
  - **Invalid** (I): NOOP

Write Buffers

- To avoid blocking the processor while a write-back is taking place, a write-buffer can be employed.
- Care must be taken that memory fetches don't occur, meanwhile, from lines that are also in the write buffer on their way back to memory.

Semantics

- Cache coherency protocols, etc. try to preserve a semantics of memory access.
- Typically they want single loads and stores to look "atomic" or "indivisible" so that the programming model is as near as possible to a theoretical MIMD ideal.

Locking, Critical Sections

- It is often necessary to have atomicity at larger grains than single reads and writes.
- Example is attaining exclusive access to some critical data structure.

Two types of locking

- Busy-waiting: processor keeps "spinning" while waiting for processor holding the lock to unlock
- Non-busy-waiting: processor blocks, turning over itself to a different process, until the lock is unlocked
- Typically the latter still entails a little busy-waiting just to access the ready-queue and waiting list of processors.

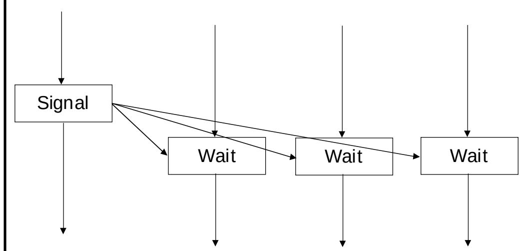
Synchronization in General

- Locking is a form of synchronization
- There are also varieties of locking:
  - Exclusive only
  - Shared-read, Exclusive-write
  - Etc.
- Other forms include "signal synchronization": one process waiting for another, as if the latter were writing data need by the former.

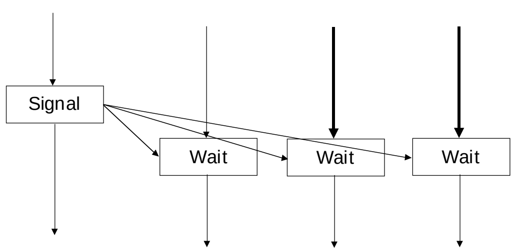
Signal Synchronization

- Different from mutual exclusion: asymmetric
- One-to-one: For each posting of an event, there is one wake-up.
- “Avalanche”: For a single posting of an event, there is an arbitrary number of wakes (all processes on the queue wakeup).

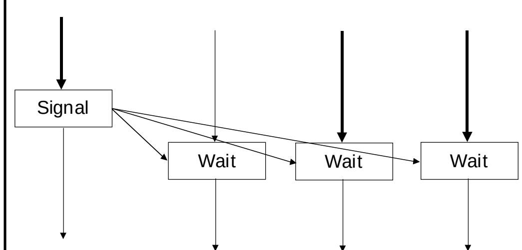
1-1 Signal Synchronization



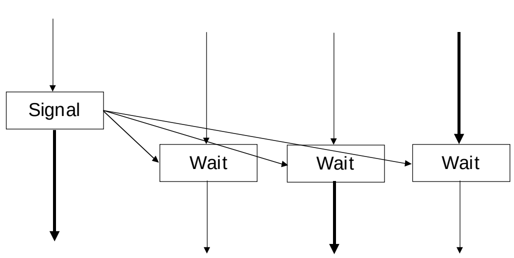
1-1 Signal Synchronization



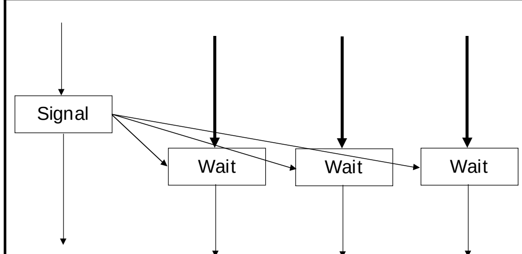
1-1 Signal Synchronization



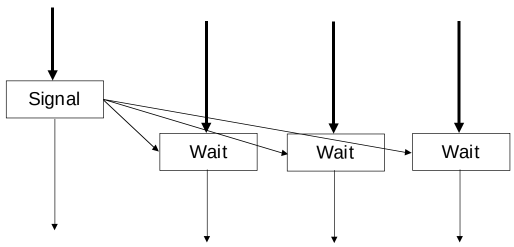
1-1 Signal Synchronization



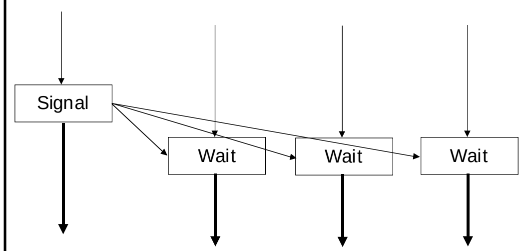
Avalanche Signal Synchronization



Avalanche Signal Synchronization



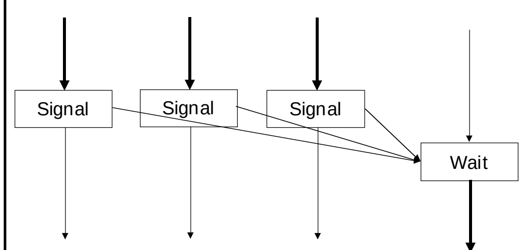
Avalanche Signal Synchronization



Multi-Join Synchronization

- The opposite case of avalanche occurs with **n-way join synchronization**: n processes have to post before the waiting process or processes proceed.
- This occurs in barriers, for example.

Multi-Join Synchronization



Threads vs. Processes

- Typically processes connote heavyweight things, threads lightweight ones
- Processes, e.g. in UNIX, contain much baggage:
  - page table
  - file descriptor table
  - processor state
  - resource tables, etc.

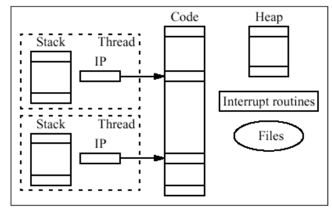
Threads vs. Processes

- Threads concentrate only on the processor state
- Consequently threads can be switched much more quickly
- This provides opportunities of latency-hiding for memory access and i/o.

Threads vs. Processes

- Threads typically share logical memory within a process.
- Processes typically do not share logical memory, except for special shareable segments.
- shmalloc = “shared memory allocate”, kind of an after-thought

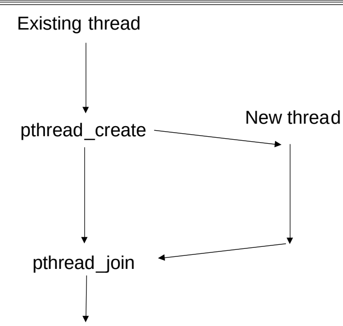
Threads within one Process



Pthreads (Posix Threads)

- Posix = an API standard, for a variety of system aspects (threads, real-time, etc.)
- Posix = “Portable UNIX”

Thread Creation and Joining



pthread_create and _exit

- pthread_create(pthread_t &tid, // thread id
  NULL, // attributes
  (void*)threadCode(void*), // code
  (void*) parameter); // params
- creates new pthread running threadCode; parameter is passed to threadCode
- pthread_exit((void*) value)
- terminates thread, passing value if joined to another thread
- pthread_join(pthread_t tid, // thread id
  (void**) result);

pthread_exit

- pthread_exit((void*) value)
- terminates thread, passing value if joined to another thread
- Note: storage for result must be allocated dynamically or outside of the thread code.

pthread_join

- pthread_join(pthread_t tid, // thread id
  (void**) result);

- waits for thread tid, result is that sent by _exit

pthread1.c example

```
struct package
{
char* msg;
};

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
  }
```

pthread1.c example

```
struct package
{
char* msg;
};

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
  }
```

```
int main(int argc, char** argv)
  {
  struct package arg1, arg2;
  char *result;
  pthread_t tid1, tid2;

  arg1.msg = "thread1";
  arg2.msg = "thread2";

  pthread_create(&tid1, NULL, threadCode, &arg1);
  pthread_create(&tid2, NULL, threadCode, &arg2);

  printf("Hello from main.\n");

  pthread_join(tid1, (void*)&result);
  printf("Thread 1 joined, result is %s.\n", result);

  pthread_join(tid2, (void*)&result);
  printf("Thread 2 joined, result is %s.\n", result);
  }
```

pthread1.c example

```
struct package
{
char* msg;
};

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
  }
```

output

```
Hello from main.
Hello from thread1.
Hello from thread2.
Thread 1 joined, result is thread1.
Thread 2 joined, result is thread2.
```

```
int main(int argc, char** argv)
  {
  struct package arg1, arg2;
  char *result;
  pthread_t tid1, tid2;

  arg1.msg = "thread1";
  arg2.msg = "thread2";

  pthread_create(&tid1, NULL, threadCode, &arg1);
  pthread_create(&tid2, NULL, threadCode, &arg2);

  printf("Hello from main.\n");

  pthread_join(tid1, (void*)&result);
  printf("Thread 1 joined, result is %s.\n", result);

  pthread_join(tid2, (void*)&result);
  printf("Thread 2 joined, result is %s.\n", result);
  }
```

Exercise

- Describe how you would implement matrix multiply using pthreads.

Thread Safety

- Some library routines might not be “thread safe”.
- This is typically because they are not “reentrant”, i.e. they assume certain fixed memory locations rather than allocate all of their storage individually.

Thread Locking (non-busy)

```
// global
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
…

// in competing threads
pthread_mutex_lock(&mutex);

… critical section …

pthread_mutex_unlock(&mutex);
```

pthread2.c example

```
struct package
{
char* msg;
pthread_mutex_t* mutex;
};


/* Using a mutex below, we should never see the hello and goodbye of two
 * threads interleaved.
 */

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  pthread_mutex_lock(realArg->mutex);
  printf("Hello from %s.\n", realArg->msg);
  sleep(1);
  printf("Goodbye from %s.\n", realArg->msg);
  pthread_mutex_unlock(realArg->mutex);
  pthread_exit(realArg->msg);
  }
```

pthread2.c example

```
int main(int argc, char** argv)
  {
  pthread_mutex_t mutex1;
  struct package arg1, arg2;
  char *result;
  pthread_t tid1, tid2;

  pthread_mutex_init(&mutex1, NULL);
  arg1.msg = "thread1";
  arg2.msg = "thread2";
  arg1.mutex = &mutex1;
  arg2.mutex = &mutex1;        // one mutex is shared with both threads

  pthread_create(&tid1, NULL, threadCode, &arg1);
  pthread_create(&tid2, NULL, threadCode, &arg2);

  pthread_mutex_lock(&mutex1);
  printf("Hello from main.\n");
  sleep(1);
  printf("Goodbye from main.\n");
  pthread_mutex_unlock(&mutex1);

  pthread_join(tid1, (void*)&result);
  printf("Thread 1 joined, result is %s.\n", result);

  pthread_join(tid2, (void*)&result);
  printf("Thread 2 joined, result is %s.\n", result);
  }
```

output

```
Hello from main.
Goodbye from main.
Hello from thread1.
Goodbye from thread1.
Hello from thread2.
Thread 1 joined, result is thread1.
Goodbye from thread2.
Thread 2 joined, result is thread2.
```

Condition Variables

- Condition variables allow one thread to signal another.

Condition Variables

```
// global
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
…

// in separate threads
pthread_cond_wait(&cond, &mutex);

pthread_cond_signal(&cond);    // 1-1 signaling
pthread_cond_broadcast(&cond); // avalanche
```

Condition Variables

```
// global
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
…

// in separate threads
pthread_cond_wait(&cond, &mutex);

pthread_cond_signal(&cond);    // 1-1 signaling
pthread_cond_broadcast(&cond); // avalanche
```

Why is this here?

pthread3.c example

```
void* threadCode( void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  if( !strcmp(realArg->msg, "thread1") )
    {
    sleep(1);
    printf("Signalling in %s.\n", realArg->msg);
    pthread_cond_signal(realArg->cond);
    }
  else
    {
    printf("Waiting in %s.\n", realArg->msg);
    pthread_cond_wait(realArg->cond, realArg->mutex);
    printf("No longer waiting in %s.\n", realArg->msg);
    sleep(1);
    }
  printf("Goodbye from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
```

```
struct package
{
char* msg;
pthread_cond_t* cond;
pthread_mutex_t* mutex;
};
```

pthread3.c example

```
int main(int argc, char** argv)
  {
  pthread_cond_t cond1;
  pthread_mutex_t mutex1;
  struct package arg1, arg2;
  char *result;
  pthread_t tid1, tid2;

  pthread_cond_init(&cond1, NULL);
  arg1.msg = "thread1";
  arg2.msg = "thread2";
  arg1.cond = &cond1;
  arg2.cond = &cond1;          // one cond is shared with both threads
  arg1.mutex = &mutex1;
  arg2.mutex = &mutex1;        // one mutex is shared with both threads

  pthread_create(&tid1, NULL, threadCode, &arg1);
  pthread_create(&tid2, NULL, threadCode, &arg2);

  printf("Hello from main.\n");
  sleep(1);
  printf("Goodbye from main.\n");

  pthread_join(tid1, (void*)&result);
  printf("Thread 1 joined, result is %s.\n", result);

  pthread_join(tid2, (void*)&result);
  printf("Thread 2 joined, result is %s.\n", result);
  }
```

output

```
Hello from main.
Hello from thread1.
Hello from thread2.
Waiting in thread2.
Goodbye from main.
Signalling in thread1.
Goodbye from thread1.
No longer waiting in thread2.
Thread 1 joined, result is thread1.
Goodbye from thread2.
Thread 2 joined, result is thread2.
```

Major, Major Caveat

- From the man page:
  - The pthread_cond_signal() and pthread_cond_broadcast() functions have no effect if there are no threads currently blocked on cond.
- This means that a collection of threads may well exhibit time-dependent behavior when using this primitive.

Semaphores

- Semaphores are a better alternative to conditional variables
- They don't lose signals that may have occurred before the wait statement.
- Exactly one wait is enabled per every signal.
- Unfortunately, they are not part of Posix

Semaphores

- Each semaphore has an associated count, initially 0 by default. (May be set at > 0)
- Invariant:
  - count > 0 $\rightarrow$ no processes waiting
  - count = number of wait operations before blocking
- Behavior:
  - wait, or P, or down:
    if(count > 0) count--; else wait on queue;
  - signal, or V, or up:
    if(queue non-empty)
        wakeup one on queue;
    else count++;

Exercise

- Implement a semaphore data type using mutexes and conditional variables.