Parallel & Concurrent Programming:

Processes & Threads

Emery Berger
CMPSCI 691W - Spring 2006

Outline

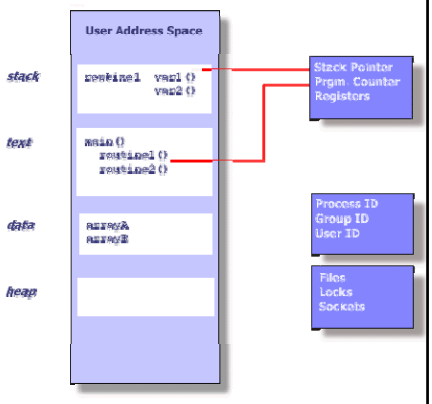
- Processes
- Threads
- Basic synchronization
- Bake-off

Processes vs. Threads…

- Both useful for parallel programming & concurrency
  - Hide latency
  - Maximize CPU utilization
  - Handle multiple, asynchronous events
- **But:** different programming styles, performance characteristics, and more

Processes

- **Process:** execution context (PC, registers) + address space, files, etc.
- Basic unit of execution

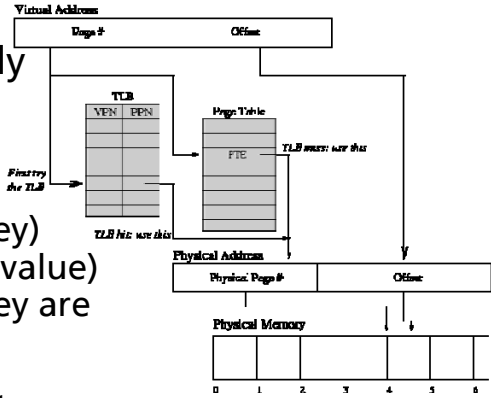


Process API

- **UNIX:**
  - **`fork()`** – create copy of current process
    - Different return value
    - **Copy-on-write**
  - **`exec()`** – replace process w/ executable
- **Windows:**
  - **`CreateProcess (…)`**
    - 10 arguments

Translation Lookaside Buffer

- TLB: fast, fully associative memory
  - Stores page numbers (key) and frame (value) in which they are stored
- Copy-on-write: **protect** pages, copy on first write



Processes Example

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
main() {
  int parentID = getpid();    /* ID of this process */
  char  prgname[1024];
  gets(prgname); /* read the name of program we want to start */
  int cid = fork();
  if(cid == 0) {  /* I'm the child process */
    execlp( prgname, prgname, 0);  /* Load the program */
    /* If the program named prgname can be started, we never get
   to this line,  because the child program is replaced by prgname */
    printf("I didn't find program %s\n", prgname);
  } else { /* I'm the parent process */
    sleep (1);  /*  Give my child time to start. */
    waitpid(cid, 0, 0);  /*  Wait for my child to terminate. */
    printf("Program %s finished\n", prgname);
} }
```

Communication

- Processes:
  - Input = state before **`fork()`**
  - Output = return value
    - argument to **`exit()`**
- **But:** how can processes communicate *during* execution?

IPC

- **signals**
  - Send & receive ints
  - Not terribly useful for parallel or concurrent programming
- **pipes**
  - Communication channels – easy & fast
  - Just like UNIX command line

Pipe example

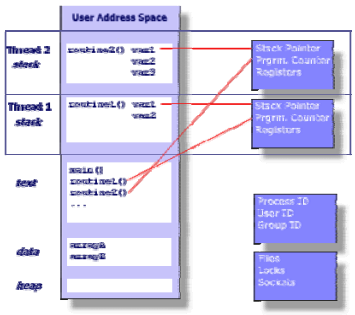
```
int main() {
  int pfds[2];
  pipe(pfds);
  if (!fork()) {
    close(1);       /* close normal stdout */
    dup(pfds[1]); /* make stdout same as pfds[1] */
    close(pfds[0]); /* we don't need this */
    execlp("ls", "ls", NULL);
  } else {
    close(0);       /* close normal stdin */
    dup(pfds[0]); /* make stdin same as pfds[0] */
    close(pfds[1]); /* we don't need this */
    execlp("wc", "wc", "-l", NULL);
  }
}
```

IPC, continued

- **sockets**
  - Explicit message passing
  - \+ Can *distribute* processes anywhere
- **shmem**
  - Best not spoken of…
- **mmap** (common hack)
  - All processes map same file into fixed memory location
  - Objects in region shared across processes
  - Use `flock()` to synchronize

Threads

- **Processes** - everything in distinct address space
- **Threads** – *same* address space (& files, sockets, etc.)



Threads API

- **UNIX (POSIX):**
  - **`pthread_create()`** – start separate *thread* executing function
  - **`pthread_join()`** – wait for thread to complete
- **Windows:**
  - **`CreateThread (…)`**
    - only 6 arguments!

Threads example

```
#include <pthread.h>
void * run (void * d) {
  int q = ((int) d);
  int v = 0;
  for (int i = 0; i < q; i++) {
    v = v + expensiveComputation(i);
  }
  return (void *) v;
}
main() {
  pthread_t t1, t2;
  int r1, r2;
  pthread_create (&t1, run, 100);
  pthread_create (&t2, run, 100);
  pthread_wait (&t1, (void *) &r1);
  pthread_wait (&t2, (void *) &r2);
  printf ("r1 = %d, r2 = %d\n", r1, r2);
}
```

Communication

- In threads, everything shared except: *stacks, registers & thread-specific* data
  - **Old way**:
    - `pthread_setspecific`
    - `pthread_getspecific`
  - **New way**: `__thread`
    - `static __thread int x;`
    - Easier in Java…
- Updates of shared state must be *synchronized*

Basic synchronization

- Mutual exclusion *locks*
  - Only one thread in *critical section*

```
pthread_mutex_lock (&l);
update data; /* critical section */
pthread_mutex_unlock (&l);
```

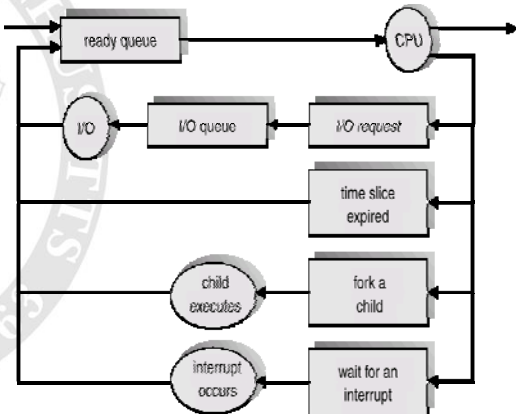
Pthreads API

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

Bake-off

- **Processes** or **threads**?
  - Performance
  - Flexibility / Ease-of-use
  - Robustness

Scheduling



Context Switch Cost

- **Threads** – much cheaper
  - Stash registers, PC ("IP"), stack pointer
- **Processes:**
  - Same as threads *plus* –
  - Process context
  - **TLB shootdown**

⇨ Process switches more expensive, or require long quanta

Flexibility / Ease-of-use

- **Processes** – more flexible
  - \+ Easy to spawn remotely
  - \+ Can communicate via sockets = can be distributed across cluster / Internet
  - \- Requires explicit communication or risky hackery
- **Threads**
  - Communicate through memory – must be on same machine
  - \- Require *thread-safe* code

Robustness

- **Processes** – far more robust
  - Processes *isolated* from other processes
    - Process dies) no effect
  - Apache 1.x
- **Threads:**
  - If one thread crashes (e.g., derefs NULL), whole process terminates
  - Then there's the stack size problem
  - Apache 2.x…

Next time

- Advanced synchronization