CS 25200: Systems Programming

Lecture 20: Threads and Synchronization

Prof. Turkstra

Lecture 20

- Threads
- Too much milk
- Mutual exclusion
- Semaphores

Creating threads

- POSIX
  pthread_create(&thr_id, attr, func, arg);
- Solaris
  thr_create(stack, stack_size, func, arg, flags, &thr_id)
- Windows
  CreateThread(attr, stack_size, func, arg, flags, &thr_id)

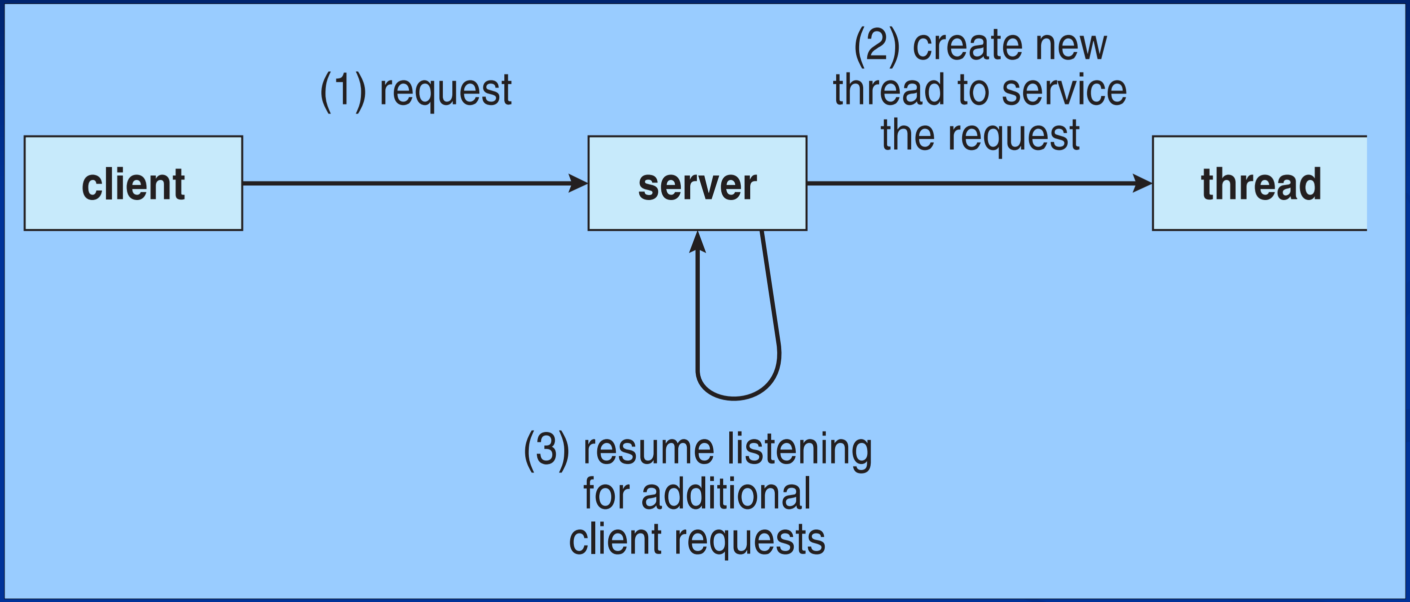
Remember

- Every thread has its own…
  - PC – program counter
  - Registers
  - State
  - Stack
- Process table entry manages this information for each thread

Why?

- Responsiveness – can continue executing if part of the process is blocked
  - Especially important for UIs
- Resource sharing – threads share resources automatically
  - Easier than explicit shared memory and message passing
- Economy – cheaper than process creation
  - Context switch overhead also lower
- Scalability – can leverage multiprocessor architectures

Concurrent server

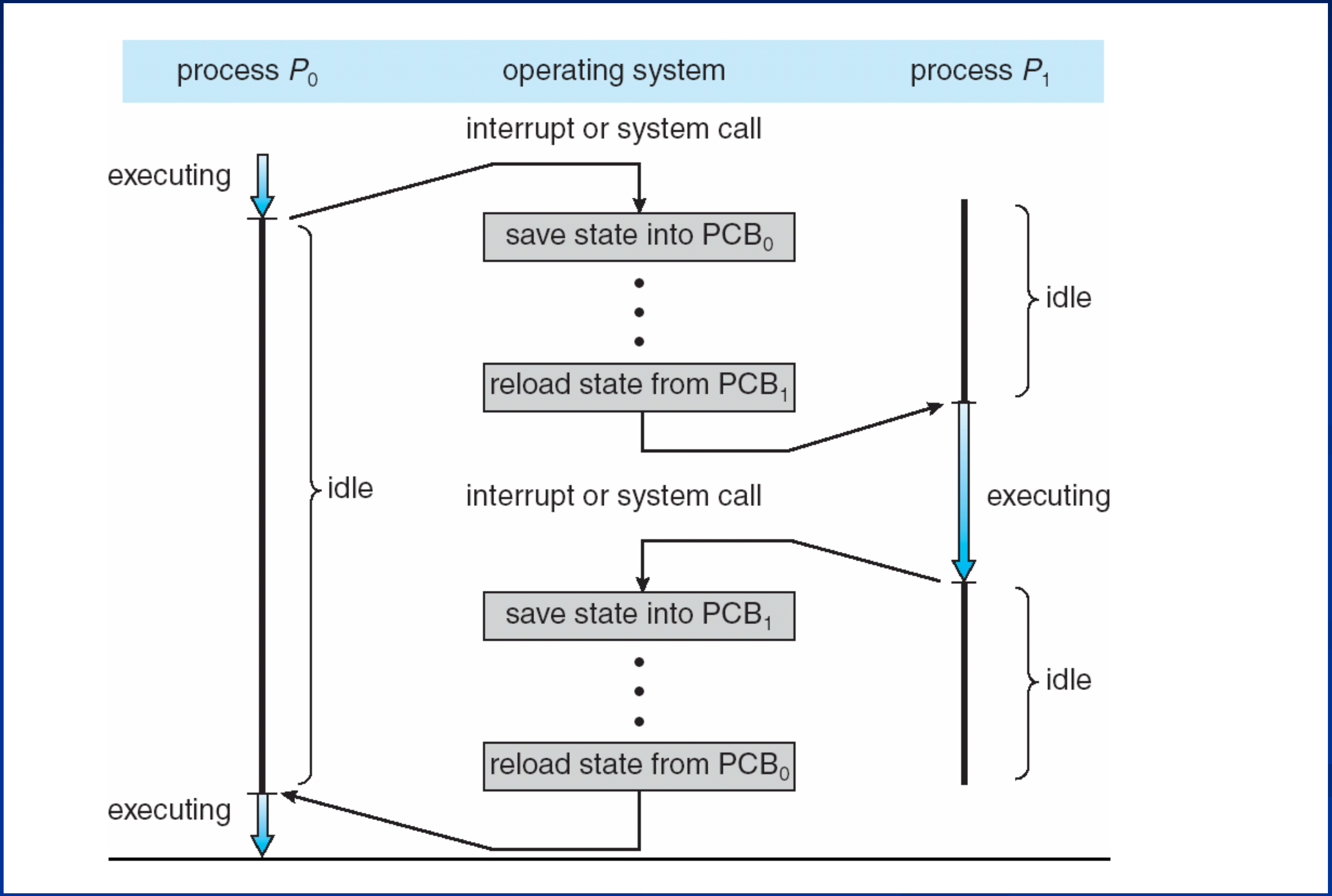


- Suppose server is single threaded
  - What happens with slow clients (e.g., modems)?

Synchronization

- Instructions are guaranteed to execute in order within a single thread
  - Or at least appear to
- No ordering guaranteed among multiple threads
  - Threads may be interrupted at any time
  - Scheduling (“time slicing”)
- So?

Context switch



Too much milk

Terminology

- Critical section: section of code or collection of operations in which only one thread may be executing at a time
- Mutual exclusion: the property that exactly one thread is doing a certain thing at one time

What do we need?

- A locking mechanism
- Lock before using
- Unlock after done
- Wait if locked

More milk

Count example

```
#include <stdio.h>
#include <pthread.h>

int count;

void inc(long n) {
  for (int i = 0; i < n; i++) {
    count++;
  }
}

void dec(long n) {
  for (int i = 0; i < n; i++) {
    count--;
  }
}
```

Count example

```
int main(int argc, char **argv) {
  long n = 10000000;
  pthread_t t1;
  pthread_t t2;

  pthread_create(&t1, NULL, (void * (*)(void *)) inc, (void *) n);
  pthread_create(&t2, NULL, (void * (*)(void *)) dec, (void *) n);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("%d\n", count);
  return 0;
}
```

Race condition

- count++/-- statement is not atomic
  - load C
  - add one to C
  - store C
- Interleaving of thread statements impacts the result
  - Non-deterministic

Non-determinism

- Instruction execution order among separate threads depends on many things
  - Threading implementation
  - Operating system (scheduling, preempting)
  - Hardware interrupts
- This occurs even on single CPU, single core systems

Synchronization

- It is the programmer's job to anticipate all possible orderings and protect against errors
- Concurrency and synchronization is hard
  - Sometimes the overhead of implementing and debugging a concurrent program is not worth it
    - Therac-25
  - Safety critical systems should always have hardware interlocks

Purdue Trivia

- Orville Redenbacher graduated from Purdue in 1928 with a degree in Agronomy
  - Marched Tuba in the AAMB
  - Also on the track team
  - Worked for the exponent
  - Honorary doctorate in 1988
- Mural in PMU basement includes him

Criteria

- Musts
  - Processes not in critical section should not block others
  - No one waits forever
  - Multi-processor friendly
- Desirable
  - Fairness – everyone eventually gets into the critical section
  - Efficient – don't waste resources (no busy waiting)
  - Simple – symmetric code, easy to use
    - Like bracketing

Processes and mutual exclusion

- Always lock before manipulating shared memory
- Always unlock after manipulating shared memory
- Do not lock again if already locked
- Do not unlock if not locked by you
- Do not spend large amounts of time in critical section

Atomic

- Appears to the entire system as occurring all at once without interruption
  - No interrupts
  - No signals
  - No concurrent processes or threads

Semaphore

- Synchronization variable that takes on positive integer values
  - Dijkstra, 60s
- Two operations:

  P(semaphore): atomic operation waits for semaphore > 0, then decrements by one
  - "Proberen" in Dutch
- V(sempahore): atomic operation increments by one
  - "Verhogen"

Pseudo code

```
P(S)
wait(S) {
  while (S < = 0);
  S--;
}
```

```
V(S)
signal(S) {
  S++;
}
```

- Done atomically
- Not usually in hardware – implementation later

Binary semaphores

- Often called a mutex or lock
- Semaphore that takes on values of 0 and 1 only
- Too much milk?

```
P(milkSemaphore)
if (!milk)
  buy milk;
V(milkSemaphore)
```

Properties

- Machine-independent
- Simple
- Works with many processes
- Can have different critical sections with different semaphores
- Can acquire many resources simultaneously (multiple P's)
- Can permit multiple processes to enter critical section at once

Usage

- Mutual exclusion
  - One process is accessing critical section at a time
  - What about separate groups of data that need to be accessed independently
- Condition synchronization
  - Permit processes to wait for something
  - What if disparate groups of processes want to wait for unrelated events?

Fixing inc.c

Implementation

- Uniprocessor solution: disable interrupts!

```
typedef struct {
  int count;
  queue q;
} semaphore;
```

P

```
void P(semaphore S) {
  disable interrupts;
  if (s->count > 0) {
    s->count--;
    enable interrupts;
    return;
  }
  add(s->q, current_thread);
  sleep(); // re-dispatch
  enable interrupts;
  return;
}
```

V

```
void V(semaphore S) {
  disable interrupts;
  if (isEmpty(s->q)) {
    s->count++;
  } else {
    thread = removeFirst(s->q);
    wakeup(thread); // put thread on ready q
  }
  enable interrupts;
  return;
}
```

Multiprocessor?

- Cannot just turn off interrupts
  - Doesn't prevent other processors from accessing shared memory
- Turn off other processors?
  - Bad :-(
- Use atomic read and write?
  - Needs to be atomic across all processors!
- Big research area for a long time

Test-and-set (IBM)

- Atomic read-modify-write instruction
- TAS – on most CISC architectures
- Semantics:
  - Set value to k, but return old value
  - k = 1 → binary semaphore

```
int lock;
while (TAS(&lock, 1) != 0);
<critical section>
lock = 0;
```

TAS

- Implemented by memory hardware or CPU refusing to relinquish bus access
- Still have to disable interrupts on current core
- Why?

RISC Mechanism

- Load-linked
  - ldl – loads a word from memory and sets per-processor flag associated with that word (in cache)
  - Store operation to same location (by any processor) resets all processors’ flags for that word
- Store-conditionally
  - stc – stores word iff flag still set, indicates success or failure

```
int lock;
while ((ldl(&lock) != 0) || !stc(&lock, 1));
<critical section>
lock = 0;
```

Questions?