pthreads II

CS 241

Oct. 2, 2013

```
void *compute(void *ptr) {
  char *s = (char *)ptr;
  printf("Line length: %d\n", strlen(s));
  return NULL;
}

void main() {
  size_t line_size = 100;
  char *line = malloc(line_size);
  FILE *file = fopen("strings.txt", "r");



  while ( getline(&line, &line_size, file) != -1 ) {
    pthread_t tid;
    pthread_create(&tid, NULL, compute, line);
  }


  close(file);
}
```

```
void *compute(void *ptr) {
  char *s = (char *)ptr;
  printf("Line length: %d\n", strlen(s));
  return NULL;
}

void main() {
  size_t line_size = 100;   int i;
  char *line = malloc(line_size);
  FILE *file = fopen("strings.txt", "r");

  pthread_t tid[100];   int ct = 0;

  while ( getline(&line, &line_size, file) != -1 ) {
     pthread_create(&tid[ct], NULL, compute, line);
     ct++;
  }

  for(i = 0; i < ct; i++)
     pthread_join(tid[i], NULL);

  close(file);
}
```

```
void *compute(void *ptr) {
  char *s = (char *)ptr;
  printf("Line length: %d\n", strlen(s));
  return NULL;
}

void main() {
  size_t line_size = 100;   int i;
  FILE *file = fopen("strings.txt", "r");

  pthread_t tid[100];   int ct = 0;
  char *lines[100];     size_t lines_len[100];
  for (i = 0; i < 100; i++)
  {  lines[i] = NULL;  lines_len[i] = 0; }

  while ( getline(&lines[ct], &lines_len[ct], file) != -1 ) {
     pthread_create(&tid[ct], NULL, compute, lines[ct]);
     ct++;
  }

  for(i = 0; i < ct; i++)
     pthread_join(tid[i], NULL);

  close(file);
}
```

Thread Safe

- A function is **thread safe** if and only if the function can safely be called by multiple threads at the same time.

- A function is **NOT** thread safe if it stores some state in a global or static variable without synchronization.

Example: strtok()

```
char *strtok ( char * str, const char * delimiters );

char *token;
char *s = malloc(100);
strcpy(s, "Computer Science 241");

token = strtok(s, " ");

token = strtok(NULL, " ");

token = strtok(NULL, " ");

token = strtok(NULL, " ");
```

Example: strtok()

```
char *strtok ( char * str, const char * delimiters );
```

```
char *token;
char *s = malloc(100);
strcpy(s, "Computer Science 241");
token = strtok(s, " ");
token = strtok(NULL, " ");
```

```
char *token;
char *s = malloc(100);
strcpy(s, "Hello World");
token = strtok(s, " ");
token = strtok(NULL, " ");
token = strtok(NULL, " ");
```

```
token = strtok(NULL, " ");
token = strtok(NULL, " ");
```

Race Condition

- A **race condition** exists when the output is dependent on the sequence or timing of other uncontrollable events.

```
int ct = 0;
int X = 10000000;

void *up(void *ptr) {
  int i;
  for (i = 0; i < X; i++)
    ct++;
}

void *down(void *ptr) {
  int i;
  for (i = 0; i < X; i++)
    ct--;
}

void main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, up, NULL);
  pthread_create(&t1, NULL, down, NULL);


  printf("Count: %d\n", ct);
}
```

Race Condition

Speedup

- One of the greatest advantages of parallelism is to speed up computation. We formally define this speedup as: $S_P = \frac{T_1}{T_P}$
  - **P**: The number of processors
  - **S_P**: The speedup for a given number of processors
  - **T_1**: The execution time of a sequential algorithm
  - **T_P**: The execution time of a parallel algorithm on P processors
- **Ideal Speedup**: